Package ‘qlcData’

July 22, 2025

Type Package

Title Processing Data for Quantitative Language Comparison

Description Functionality to read, recode, and transcode data as used in quantitative language comparison, specifically to deal with multilingual orthographic variation (Moran & Cysouw (2018) <doi:10.5281/zenodo.1296780>) and with the recoding of nominal data.

Version 0.3

Date 2024-06-07

License GPL-3

Encoding UTF-8

Depends R (>= 3.5.0)

Imports stringi (>= 0.2-5), yaml (>= 2.1.11), shiny, docopt, data.tree, phytools, ape

Suggests knitr, rmarkdown, DiagrammeR

VignetteBuilder knitr

LazyData true

NeedsCompilation no

Author Michael Cysouw [aut, cre] (ORCID: <`https://orcid.org/0000-0003-3168-4946`>)

Maintainer Michael Cysouw <`cysouw@mac.com`>

Repository CRAN

Date/Publication 2024-06-10 17:10:01 UTC

Contents

qlcData-package 2
.expandValues 3
asPhylo 3
getTree 5
glottolog 7

join_align 9
pass_align 10
read_align 11
recode 12
tokenize 14
write.profile 18
write.recoding 21

Index **24**

`qlcData-package` *Processing data for quantitative language comparison (QLC)*

Description

The package offers various functions to read, transcode and process data. There are many different function to read in data. Also a general framework to recode nominal data is included. Further, there is a general approach to describe orthographic systems through so-called Orthography Profiles. It offers functions to write such profiles based on some actual written text, and to test and correct profiles given concrete data. The main end-use is to produce tokenized texts in so-called tailored grapheme clusters.

Details

Package:	qlcData
Type:	Package
Version:	0.3
Date:	2024-06-07
License:	GPL-3

Various functions to read specific data formats of QLC are documented in `read_align`, `read.profile`, `read.recoding`.

The `recode` function allows for an easy and transparent way to specify a recoding of an existing nominal dataset. The specification of the recoding-decisions is preferably saved in an easily accessible YAML-file. There are utility function `write.profile` for writing and reading such files included.

For processing of strings using orthography profiles, the central function is `tokenize`. A basic sceleton for an orthography profile can be produced with `write.profile`

Author(s)

Michael Cysouw <cysouw@mac.com>

`.expandValues` *Internal helper*

Description

produce combinations of nominal variables

Usage

```
.expandValues(attributes, data, all)
```

Arguments

`attributes`	a list of attributes to be recoded. Vectors (as elements of the list) are possible to specify combinations of attributes to be recoded as a single complex attribute.
`data`	a data frame with nominal data, attributes as columns, observations as rows.
`all`	Logical: should all values be produced or only those with existing data?

Details

Just a helper.

Value

`expandValues` is an internal help function to show the various value-combinations when combining attributes.

Author(s)

Michael Cysouw

`asPhylo` *Convert Glottolog trees to phylo format*

Description

Convenience version of conversion to phylo format of the ape package, used throughout packages for phylogenetic methods. This conversion offers various options to tweak the branch lenghts of Glottolog trees (which do not have any inherent branch lengths). Currently very slow for large trees!

Usage

```
asPhylo(tree, height = 100, fixed.branches = NULL, long.root = NULL,
  multi.allow = FALSE, quick = FALSE)
```

Arguments

`tree`	(selection of) Glottolog data, preferably extracted with `getTree`. If the selection is not logically a tree, then it will lead to errors.
`height`	Height of the tree. By default all nodes in the tree will be simply equally spaced on this height in an ultrametric fashion, i.e. all leaves will be at height zero.
`fixed.branches`	Alternatively to `height`, specify a fixed length for all branches. Note that this happens before all singleton nodes will be removed, so branches with singleton nodes will become as long as the amount of singleton nodes on it.
`long.root`	The Glottolog version included in this packages includes a 'World' root and six area nodes below this root. These nodes are not strictly speaking genealogical nodes, though they are often practical for phylogenitic inference. Specify the length of these nodes here.
`multi.allow`	Allow multichotomies in tree
`quick`	Conversion when quick approach is used

Details

The `phylo` format is a widely used format for phylogenetic inference. However, many methods depend crucially on branch lengths. As linguistic trees mostly do not have branch lenghts, this conversion offers a few options to tweak branch lengths.

Note that phylo trees do not allow singleton nodes, so they are removed internally (by `collapse.singles`). The tree is also forced to be binary (by `multi2di`), which is also expected in many phylogenetic analysis.

The order of length assignment is: reduce (in getTree) » height » fixed length » long root » collapse singles, which can be used to lead to different results. Also the option `reduce` in `getTree` can be used to influence the branch lenghts (see Examples below).

The collapsing of singleton internal nodes after branch length leads to an interesting effect that branch lengths are somehow proportional to the number of internal diversity of the tree, which might make sense as a proxy to branch lengths. Something like `reduce = F`, `fixed.branches = 1`, `long.root = 10` seem promising.

Value

An object of class `phylo`.

Note

Depends internally on `ToNewick` which is currently very slow for large trees.

Author(s)

Michael Cysouw <cysouw@mac.com

Examples

```
# many different effects can be achieved by combining options

isoCodes <- c("deu", "eng", "swe", "swh", "xho", "fin")
treeWithInternal <- getTree(isoCodes, reduce = FALSE)
treeNoInternal <- getTree(isoCodes, reduce = TRUE)


# to understand the influence of the option 'reduce' also check
library(data.tree)
plot(FromDataFrameNetwork(treeWithInternal))
plot(FromDataFrameNetwork(treeNoInternal))


library(ape)
oldpar <- par("mfcol")
par(mfcol = c(2,3))

phylo <- asPhylo(treeWithInternal, height = 20)
plot(phylo, main = "reduce = FALSE\nheight = 20", cex = 1)
edgelabels(round(phylo$edge.length), cex = 1.5)

phylo <- asPhylo(treeNoInternal, height = 20)
plot(phylo, main = "reduce = TRUE\nheight = 20", cex = 1)
edgelabels(round(phylo$edge.length), cex = 1.5)

phylo <- asPhylo(treeWithInternal, fixed.branches = 1)
plot(phylo, main = "reduce = FALSE\nfixed.branches = 1", cex = 1)
edgelabels(round(phylo$edge.length), cex = 1.5)

phylo <- asPhylo(treeNoInternal, fixed.branches = 1)
plot(phylo, main = "reduce = TRUE\nfixed.branches = 1", cex = 1)
edgelabels(round(phylo$edge.length), cex = 1.5)

phylo <- asPhylo(treeWithInternal, fixed.branches = 1, long.root = 10)
plot(phylo, main = "reduce = FALSE\nfixed.branches = 1, long.root = 10", cex = 1)
edgelabels(round(phylo$edge.length), cex = 1.5)

phylo <- asPhylo(treeNoInternal, fixed.branches = 1, long.root = 10)
plot(phylo, main = "reduce = TRUE\nfixed.branches = 1, long.root = 10", cex = 1)
edgelabels(round(phylo$edge.length), cex = 1.5)

par(mfcol = oldpar)
```

getTree *Extract parts out of the full Glottolog 2016 tree*

Description

getTree is a convenience function to extract parts from the Glottolog 2016 data as provided here in `glottolog`.

Usage

```
getTree(up = NULL, kind = "iso", down = NULL, reduce = FALSE)
```

Arguments

`up`	a vector of names from which to extract the tree upwards. Can be names from Glottolog, ISO 639-3 codes, WALS codes or glottocodes. Default settings expect ISO 639-3 codes.
`kind`	what kind of names are specified in `up`? Choose one of `iso`, `wals`, `glottocode` or `name` for full names from Glottolog.
`down`	a vector of family names from Glottolog from which to extract the tree downwards. For consistency, a node "World" as added on top, linking separate families. Specifying families that are part of other families in `down` will lead to a warning, but any overlap will be gracefully removed.
`reduce`	remove all nodes in the tree that do not branch.

Details

Specifying both `up` and `down` will extract only the intersection of the two. So any name in `up` that lies outside any family in `down` will be discarded.

Value

Returns a data.frame with the relevant lines from the full glottolog data.

Note

This function is hard-coded to only use the data as available in `glottolog`.

Author(s)

Michael Cysouw <cysouw@mac.com

See Also

`asPhylo` for tweaking branch lengths in phylo conversion.

Examples

```
# use getTree() to select genealogical parts of the data
data(glottolog)

( aalawa <- getTree(up = "aala1237", kind = "glottocode") )
( kandas <- getTree(down = "Kandas-Duke of York") )
( treeFull <- getTree(up = c("deu", "eng", "ind", "cha"), kind = "iso") )
( treeReduced <- getTree(up = c("deu", "eng", "ind", "cha"), kind = "iso", reduce = TRUE) )


# use FromDataFrameNetwork() to visualize the tree
# and export it into various other tree formats in R

library(data.tree)
treeF <- FromDataFrameNetwork(treeFull)
treeR <- FromDataFrameNetwork(treeReduced)

plot(treeF)
plot(treeR)

# turn into phylo format from library 'ape'
t <- as.phylo.Node(treeR)
plot(t)

# turn into dendrogram
t <- as.dendrogram(treeF)
plot(t, center = TRUE)
```

glottolog *Glottolog data from* https://glottolog.org

Description

Data from Glottolog 2016 with added WALS codes and speaker-community size. Various minor corrections and additions were performed in the preparation of the data (see Details). All stocks (i.e. largest reconstructable units) are linked to macroareas, and they are linked to a single root node calles 'World'.

Usage

```
data("glottolog")
```

Format

A data frame with 22007 observations on the following 10 variables.

`name` a character vector with the name of the entity.

`father` a character vector with the name of the direct parent entity.

`stock` a factor with the highest reconstructable unit. This column is added just for convenience, it does not add any new information.

`glottocode` a character vector with the glottocode. The same identifier is added as rownames of the data.

`iso` a character vector with ISO 639-3 language codes `wals` a character vector with WALS language codes

`type` a factor with levels `dialect`, `family` and `language`

`longitude` a numeric vector with geographic coordinates as available in the Glottolog

`latitude` a numeric vector with geographic coordinates as available in the Glottolog

`population` a numeric vector with speaker community size from an old Ethnologue version (13th Edition), licensed to the MPI-EVA in Leipzig.

Details

For Glottolog data: the names were uniquified by adding a glottocode when a name occurs more than once (typically in some cases of a language and a family having the same name). Entries classified as 'bookkeeping', 'unattested' 'artificial language', 'sign language', 'speech register' and 'unclassifiable' were removed. Links to WALS codes were added: note that about 20 links are missing, and for the non-unique links one link was chosen by data availability. Some macro codes from ISO 639-3 were added.

A level 'area' was added to the tree, separating all languages in six areas: Eurasia, Africa, Southeast Asia, Sahul, North America and South America. This is reminiscent of the proposal from Dryer (1992), though Austronesian is grouped with Southeast Asia here, because that makes more sense genealogically. Still, these nodes are surely not monophyletic! Mixed languages are not assigned to an area.

Please note that the data provided here is not identical to the online version of Glottolog, as the online version is constantly being updated! This is Glottolog 2016. Updates might be made available when they are provided for download from the website.

The format of the glottolog data might seem a bit convoluted, but by using `getTree` it is actually really easy to extract genealogical parts of the glottolog data and by using `FromDataFrameNetwork` this can be nicely plotted and turned into various tree format as used in R.

Source

Glottolog 2016 data from `https://glottolog.org`. WALS 2013 data from `https://glottolog.org`. Information on macrolanguages from `https://iso639-3.sil.org/code_tables/macrolanguage_mappings/data`. All data downloaded in March 2017. Population numbers are from the 13th edition of the Ethnologue, licenced to the MPI-EVA in Leipzig.

Examples

```
# use getTree() to select genealogical parts of the data
data(glottolog)

( aalawa <- getTree(up = "aala1237", kind = "glottocode") )
( kandas <- getTree(down = "Kandas-Duke of York") )
( treeFull <- getTree(up = c("deu", "eng", "ind", "cha"), kind = "iso") )
( treeReduced <- getTree(up = c("deu", "eng", "ind", "cha"), kind = "iso", reduce = TRUE) )

# check out areas
( areas <- glottolog[glottolog$type == "area", "name"] )
# stocks in Southeast Asia
glottolog[glottolog$father == areas[1], "name"]
```

```
# use FromDataFrameNetwork() to visualize the tree
# and export it into various other tree formats in R

library(data.tree)
treeF <- FromDataFrameNetwork(treeFull)
treeR <- FromDataFrameNetwork(treeReduced)

plot(treeF)
plot(treeR)

# turn into phylo format from library 'ape'
t <- as.phylo.Node(treeR)
plot(t)

# turn into dendrogram
t <- as.dendrogram(treeF)
plot(t, center = TRUE)
```

`join_align` *Join various multialignments into one combined dataframe*

Description

Multialignments are mostly stored in separate files per cognateset. This function can be used to bind together a list of multialignments read by read_align.

Usage

```
join_align(alignments)
```

Arguments

`alignments`	A list of objects as returned from read_align.

Details

The alignments have to be reordered for this to work properly. Also, duplicate data (i.e. multiple words from the same language) will be removed. Simply the first occurrence is retained. This is not ideal, but it is currently the best and easiest solution.

Value

The result will be a dataframe with `doculects` as rows and `alignments` as columns.

Author(s)

Michael Cysouw <cysouw@mac.com>

pass_align *Transfer alignment from one string to another*

Description

In the alignment of linguistic strings, it is often better to perform the alignment on a simplified string. This function allows to pass back the alignment from the simplified string to the original

Usage

```
pass_align(originals, alignment, sep = " ", in.gap = "-", out.gap = "-")
```

Arguments

`originals`	Vector of strings in the original form, with separators
`alignment`	Vector of simplified strings after alignment, with separators and gaps. The number of non-gap parts should match the number of parts of the originals
`sep`	Symbol used as separator between parts of the strings
`in.gap`	Symbol used as gap indicator in the alignments
`out.gap`	Symbol used as gap indicator in the output. This is useful when the gap symbol from the alignments occurs as character in the originals .

Details

Given some strings, a sound (or graphemic) alignment inserts gaps into the strings in such a way as to align the columns between different strings. We assume here an original string that is separated by `sep` into parts (segments, sounds, tailored grapheme clusters). After simplification (e.g. through `tokenize`) and alignment (currently using non-R software) a string is retuned with extra gaps inserted. The number of non-gap parts should match the original string.

Value

Vector of original strings with the gaps inserted from the aligned strings.

Note

There is a bash-executable distributed with this package (based on the `docopt` package) that let you use this function directly in a bash-terminal. The easiest way to use this executable is to softlink the executable to some directory in your bash PATH, for example `/usr/local/bin` or simply `~/bin`. To softlink the function `tokenize` to this directory, use something like the following in your bash terminal:

```
ln -is `Rscript -e 'cat(system.file("exec/pass_align", package="qlcData"))'` ~/bin
```

From within R your can also use the following (again, optionally changing the linked-to directory from `~/bin` to anything more suitable on your system):

```
file.symlink(system.file("exec/pass_align", package="qlcData"), "~/bin")
```

Author(s)

Michael Cysouw <cysouw@mac.com>

Examples

```
# make some strings with separators
l <- list(letters[1:3], letters[4:7], letters[10:15])
originals <- sapply(l, paste, collapse = " ")
cbind(originals)

# make some alignment
# note that this alignment is non-sensical!
alignment <- c("X - - - X - X", "X X - - - X X", "X X X - X X X")
cbind(alignment)

# match originals to the alignment
transferred <- pass_align(originals, alignment)
cbind(transferred)

# ========

# a slighly more interesting example
# using the bare-bones pairwise alignment from adist()
originals <- c("cute kitten class","utter tentacles")
cbind(originals)

# adist returns strings of pairwise Levenshtein operations
# "I" signals insertion
(levenshtein <- attr(adist(originals, counts = TRUE), "trafos"))

# pass alignments to original strings, show the insertions as "-" gaps
alignment <- c(levenshtein[1,2], levenshtein[2,1])
transferred <- pass_align(originals, alignment,
    sep = "", in.gap = "I", out.gap = "-")
cbind(transferred)
```

`read_align` *Reading different versions of linguistic multialignments.*

Description

Multialignments of strings are a central step for historical linguistics (quite similar to multialignments in bioinformatics). There is no consensus (yet) about the file-structure for multialignments in linguistics. Currently, this functions offers to read various flavours of multialignment, trying to harmonize the internal R-structure.

Usage

```
read_align(file, flavor)
```

Arguments

`file`	Multialignment to be read
`flavor`	Currently two flavours are implemented `"PAD"` and `"BDPA"`

Details

The flavor `"PAD"` refers to the Phonetische Atlas Deutschlands, which provides multialignments for german dialects. The flavor `"BDPA"` refers to the Benchmark Database for Phonetic Alignments.

Value

Multialignment-files often contain various different kinds of information. An attempt is made to turn the data into a list with the following elements:

`meta`	: Metadata
`align`	: The actual alignments as a dataframe. When IDs are present in the original file, they are used as rownames. Some attempt is made to add useful column names.
`doculects`	: The rows of the alignment normally are some kind of doculects ("languages", "dialects"). However, because these doculects might occur more than once (when two different, but cognate words from a languages are included) these names are not used as rownames of `$align`, but presented separately here.
`annotations`	: The columns of a multialignment can have annotations, e.g. metathesis or orthographic standard. These annotations are saved here as a dataframe with the same number of columns as the `$align` dataframe. The name of the annotation is put in the rownames.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

BDPA is available at `https://alignments.lingpy.org`. PAD is available at `https://github.com/cysouw/PAD/`

`recode` *Recoding nominal data*

Description

Nominal data ('categorical data') are data that consist of attributes, and each attribute consists of various discrete values ('types'). The different values that are distinguished in comparative linguistics are mostly open to debate, and different scholars like to make different decisions as to the definition of values. The `recode` function allows for an easy and transparent way to specify a recoding of an existing dataset.

Usage

```
recode(recoding, data = NULL)
```

Arguments

`recoding`	a `recoding` data structure, specifying the decisions of the recoding. It can also be a path to a file containing the specifications in YAML format. See Details.
`data`	a data frame with nominal data, attributes as columns, observations as rows. If nothing is provided, an attempt is made to read the data with `read.csv` from the relative path provided in `originalData` in the metadata of the recoding.

Details

Recoding nominal data is normally considered too complex to be performed purely within R. It is possible to do it completely within R, but it is proposed here to use an external YAML document to specify the decisions that are taken in the recoding. The typical process of recoding will be to use write.recoding to prepare a skeleton that allows for quick and easy YAML-specification of a recoding. Or a YAML-recoding is written manually using various shortcuts (see below), and read.recoding is used to turn it into a full-fledged recoding that can also be used to document the decisions made. The function `recode` then combines the original data with the recoding, and produces a recoded dataframe.

The `recoding data structure` in the YAML document basically consists of a list of recodings, each of which describes a new attribute, based on one or more attributes from the original data. Each new attribute is described by:

- *attribute*: the new attribute name.
- *values*: a character vector with the new value names.
- *link*: a numeric vector with length of the original number of values. Each entry specifies the number of the new value. Zero can be used for any values that should be ignored in the new attribute.
- *recodingOf*: the name(s) of the original attribute that forms the basis of the recoding. If there are multiple attributes listed, then the new attribute will be a combination of the original attributes.
- *OriginalValues*: a character vector with the value names from the original attribute. These are only added to the template to make it easier to specify the recoding. In the actual recoding the listing in this file will be ignored. It is important to keep the ordering as specified, otherwise the linking will be wrong. The ordering of the values follows the result of `levels`, which is determined by the current locale.

There is a vignette available with detailed information about the process of recoding, check `recoding nominal data`.

Value

`recode` returns a data frame with the recoded attributes

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Cysouw, Michael, Jeffrey Craig Good, Mihai Albu and Hans-Jörg Bibiko. 2005. Can GOLD "cope" with WALS? Retrofitting an ontology onto the World Atlas of Language Structures. *Proceedings of E-MELD Workshop 2005*, https://emeld.org/workshop/2005/papers/good-paper.pdf

See Also

The World Atlas of Language Structure (WALS) contains typical data that most people would very much like to recode before using for further analysis. See Cysouw et al. 2005 for a discussion of various issues surrounding the WALS data.

`tokenize` *Tokenization and transliteration of character strings based on an orthography profile*

Description

To process strings it is often very useful to tokenise them into graphemes (i.e. functional units of the orthography), and possibly replace those graphemes by other symbols to harmonize the orthographic representation of different orthographic representations (‘transcription/transliteration’). As a quick and easy way to specify, save, and document the decisions taken for the tokenization, we propose using an orthography profile.

This function is the main function to produce, test and apply orthography profiles.

Usage

```
tokenize(strings,
  profile = NULL, transliterate = NULL,
  method = "global", ordering = c("size", "context", "reverse"),
  sep = " ", sep.replace = NULL, missing = "\u2047", normalize = "NFC",
  regex = FALSE, silent = FALSE,
  file.out = NULL)
```

Arguments

`strings`	Vector of strings to the tokenized. It is also possibly to pass a filename, which will then simply be read as `scan(strings, sep = "\n", what = "character")`.
`profile`	Orthography profile specifying the graphemes for the tokenization, and possibly any replacements of the available graphemes. Can be a reference to a file or an R object. If NULL then the orthography profile will be created on the fly using the defaults of `write.profile`.

`transliterate`	Default `NULL`, meaning no transliteration is to be performed. Alternatively, specify the name of the column in the orthography profile that should be used for replacement.
`method`	Method to be used for parsing the strings into graphemes. Currently two options are implemented: `global` and `linear`. See Details for further explanation.
`ordering`	Method for ordering. Currently three different methods are implemented, which can be combined (see Details below): `size`, `context`, `reverse` and `frequency`. Use `NULL` to prevent ordering and use the top to bottom order as specified in the orthography profile.
`sep`	Separator to be inserted between graphemes. Defaults to space. This function assumes that the separator specified here does not occur in the data. If it does, unexpected things might happen. Consider removing the chosen seperator from your strings first, e.g. by using `gsub` or use the option `sep.replace`.
`sep.replace`	Sometimes, the chosen separator (see above) occurs in the strings to be parsed. This is technically not a problem, but the result might show unexpected sequences. When `sep.replace` is specified, this marking is inserted in the string at those places where the `sep` marker occurs. Typical usage in linguistics would be `sep = " ", sep.replace = "#"` adding spaces between graphemes and replacing spaces in the input string by hashes in the output string.
`missing`	Character to be inserted at transliteration when no transliteration is specified. Defaults to DOUBLE QUESTION MARK at U+2047. Change this when this character appears in the input string.
`normalize`	Which normalization to use before tokenization, defaults to "NFC". Other option is "NFD". Any other input will result in no normalisation being performed.
`regex`	Logical: when `regex = FALSE` internally the matching of graphemes is done exact, i.e. without using regular expressions. When `regex = TRUE` ICU-style regular expression (see `stringi-search-regex`) are used for all content in the profile (including the Grapheme-column!), so any reserved characters have to be escaped in the orthography profile. Specifically, add a slash "\" before any occurrence of the characters `[](){}|+*.-!?^$\` in your profile (except of course when these characters are used in their regular expression meaning). Note that this parameter also influences whether contexts should be considered in the tokenization (internally, contextual searching uses regular expressions). By default, when `regex = FALSE`, context is ignored. If `regex = TRUE` then the function checks whether there are columns called `Left` (for the left context) and `Right` (for the right context), and optionally a column called `Class` (for the specification of grapheme-classes) in the orthography profile. These are hard-coded column-names, so please adapt your orthography profile accordingly. The columns `Left` and `Right` allow for regular expression to specify context.
`silent`	Logical: by default missing characters in the strings are reported with a warning. use `silent = TRUE` to supress these warnings.
`file.out`	Filename for results to be written. No suffix should be specified, as various different files with different suffixes are produced (see Details below). When `file.out` is specified, then the data is written to disk AND the R dataframe is returned invisibly.

Details

Given a set of graphemes, there are at least two different methods to tokenize strings. The first is called `global` here: this approach takes the first grapheme, matches this grapheme globally at all places in the string, and then turns to the next string. The other approach is called `linear` here: this approach walks through the string from left to right. At the first character it looks through all graphemes whether there is any match, and then walks further to the end of the match and starts again. In some special cases these two methods can lead to different results (see Examples).

The ordering or the lines in the ortography profile is of crucial importance, and different orderings will lead to radically different results. To simply use the top to bottom ordering as specified in the profile, use `order = NULL`. Currently, there are four different ordering strategies implemented: `size`, `context`, `reverse` and `frequency`. By specifying more than one in a vector, these orderings are used to break ties, e.g. the default specification `c("size", "context", "reverse")` will first order by size, and for those with the same size, it will order by whether any context is specifed (with context coming first). For lines that are still tied (i.e. the have the same size and all either have or have no context) the order will be reversed in comparison to the order as attested in the profile. Reversing order can be useful, because hand-written profiles tend to put general rules before specific rules, which mostly should be applied in reverse order.

- `size`: order the lines in the profile by the size of the grapheme, largest first. Size is measured by number of Unicode characters after normalization as specified in the option `normalize`. For example, `é` has a size of 1 with `normalize = "NFC"`, but a size of 2 with `normalize = "NFD"`.
- `context:` order the lines by whether they have any context specified, lines with context coming first. Note that this only works when the option `regex = TRUE` is also chosen.
- `reverse:` order the lines from bottom to top.
- `frequency`: order the lines by the frequency with which they match in the specified strings before tokenization, least frequent coming first. This frequency of course depends crucially on the available strings, so it will lead to different orderings when applied to different data. Also note that this frequency is (necessarily) measured before graphemes are identified, so these ordering frequencies are not the same as the final frequencies shown in the outpur. Frequency of course also strongly differs on whether context is used for the matching through `regex = TRUE`.

Value

Without specificatino of `file.out`, the function `tokenize` will return a list of four:

`strings`	a dataframe with the original and the tokenized/transliterated strings
`profile`	a dataframe with the graphemes with added frequencies. The dataframe is ordered according to the order that resulted from the specifications in `ordering`.
`errors`	a dataframe with all original strings that contain unmatched parts.
`missing`	a dataframe with the graphemes that are missing from the original orthography profilr, as indicated in the errors. Note that the report of missing characters does currently not lead to correct results for transliterated strings.

When `file.out` is specified, these four tables will be written to three different tab-separated files (with header lines): `file_strings.tsv` for the strings, `file_profile.tsv` for the orthrography profile, `file_errors.tsv` for the strings that have unidentifyable parts, and `file_missing.tsv` for the graphemes that seem to be missing. When there is nothing missing, then no file for the missing strings is produced.

Note

When `regex = TRUE`, regular expressions are acceptable in the columns ‘Grapheme‘, ’Left’ and ’Right’. Backreferences in the transliteration column are not possible (yet). When regular expressions are allowed, all literal uses of special regex-characters have to be escaped! Any literal occurrence of the following characters has then to be preceded by a backslash `\` .

- - (U+002D, HYPHEN-MINUS)
- ! (U+0021, EXCLAMATION MARK)
- ? (U+003F, QUESTION MARK)
- . (U+002E, FULL STOP)
- ((U+0028, LEFT PARENTHESIS)
-) (U+0029, RIGHT PARENTHESIS)
- \[(U+005B, LEFT SQUARE BRACKET)
- \] (U+005D, RIGHT SQUARE BRACKET)
- { (U+007B, LEFT CURLY BRACKET)
- } (U+007D, RIGHT CURLY BRACKET)
- | (007C, VERTICAL LINE)
- * (U+002A, ASTERISK)
- \ (U+005C, REVERSE SOLIDUS)
- ^ (U+005E, CIRCUMFLEX ACCENT)
- + (U+002B, PLUS SIGN)
- $ (U+0024, DOLLAR SIGN)

Note that overlapping matching does not (yet) work with regular expressions. That means that for example "aa" is only found once in "aaa". In some special cases this might lead to problems that might have to be explicitly specified in the profile, e.g. a grapheme "aa" with a left context "a". See examples below. This problem arises because overlap is only available in literal searches `stri_opts_fixed`, but the current function uses regex-searching, which does not catch overlap `stri_opts_regex`.

Note

There is a bash-executable distributed with this package (based on the `docopt` package) that let you use this function directly in a bash-terminal. The easiest way to use this executable is to softlink the executable to some directory in your bash PATH, for example `/usr/local/bin` or simply `~/bin`. To softlink the function `tokenize` to this directory, use something like the following in your bash terminal:

```
ln -is `Rscript -e 'cat(system.file("exec/tokenize", package="qlcData"))'` ~/bin
```

From within R your can also use the following (again, optionally changing the linked-to directory from `~/bin` to anything more suitable on your system):

```
file.symlink(system.file("exec/tokenize", package="qlcData"), "~/bin")
```

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Moran & Cysouw (forthcoming)

See Also

See also `write.profile` for preparing a skeleton orthography profile.

Examples

```
# simple example with interesting warning and error reporting
# the string might look like "AABB" but it isn't...
(string <- "\u0041\u0410\u0042\u0412")
tokenize(string,c("A","B"))

# make an ad-hoc orthography profile
profile <- cbind(
    Grapheme = c("a","ä","n","ng","ch","sch"),
    Trans = c("a","e","n","N","x","sh"))
# tokenization
tokenize(c("nana", "änngschä", "ach"), profile)
# with replacements and a warning
tokenize(c("Naná", "änngschä", "ach"), profile, transliterate = "Trans")

# different results of ordering
tokenize("aaa", c("a","aa"), order = NULL)
tokenize("aaa", c("a","aa"), order = "size")

# regexmatching does not catch overlap, which can lead to wrong results
# the second example results in a warning instead of just parsing "ab bb"
# this should occur only rarely in natural language
tokenize("abbb", profile = c("ab","bb"), order = NULL)
tokenize("abbb", profile = c("ab","bb"), order = NULL, regex = TRUE)

# different parsing methods can lead to different results
# note that in natural language this is VERY unlikely to happen
tokenize("abc", c("bc","ab","a","c"), order = NULL, method = "global")$strings
tokenize("abc", c("bc","ab","a","c"), order = NULL, method = "linear")$strings
```

`write.profile` *Writing (and reading) of an orthography profile skeleton*

Description

To process strings, it is often very useful to tokenise them into graphemes (i.e. functional units of the orthography), and possibly replace those graphemes by other symbols to harmonize the orthographic representation of different orthographic representations ('transcription'). As a quick and easy way to specify, save, and document the decisions taken for the tokenization, we propose using an orthography profile.

Provided here is a function to prepare a skeleton for an orthography profile. This function takes some strings and lists detailed information on the Unicode characters in the strings.

Usage

```
write.profile(strings,
    normalize = NULL, info = TRUE, editing = FALSE, sep = NULL,
    file.out = NULL, collation.locale = NULL)

read.profile(profile)
```

Arguments

`strings`	A vector of strings on which to base the orthography profile. It is also possibly to pass a filename, which will then simply be read as `scan(strings, sep = "\n", what = "character")`.
`normalize`	Should any unicode normalization be applied before making a profile? By default, no normalization is applied, giving direct feedback on the actual encoding as observed in the strings. Other options are `NFC` and `NFD`. In combination with `sep` these options can lead to different insights into the structure of your strings (see examples below).
`info`	Add columns with Unicode information on the graphemes: Unicode code points, Unicode names, and frequency of occurrence in the input strings.
`editing`	Add empty columns for further editing of the orthography profile: left context, right context, class, and translitation. See `tokenize` for detailed information on their usage.
`sep`	separator to separate the strings. When NULL (by default), then unicode character definitions are used to split (as provided by UCI, ported to R by `stringi::stri_split_boundaries`. When `sep` is specified, strings are split by this separator. Often useful is `sep = ""` to split by unicode codepoints (see examples below).
`file.out`	Filename for writing the profile to disk. When `NULL` the profile is returned as an R dataframe consisting of strings. When `file.out` is specified (as a path to a file), then the profile is written to disk and the R dataframe is returned invisibly.
`collation.locale`	Specify to ordering to be used in writing the profile. By default it uses the ordering as specified in the current locale (check `Sys.getlocale("LC_COLLATE")`).
`profile`	An orthography profile to be read. Has to be a tab-delimited file with a header. There should be at least a column called "Grapheme".

Details

String are devided into default grapheme clusters as defined by the Unicode specification. Underlying code is due to the UCI as ported to R in the `stringi` package.

Value

A dataframe with strings representing a skeleton of an orthography profile.

Note

There is a bash-executable distributed with this package (based on the `docopt` package) that let you use this function directly in a bash-terminal. The easiest way to use this executable is to softlink the executable to some directory in your bash PATH, for example `/usr/local/bin` or simply `~/bin`. To softlink the function `tokenize` to this directory, use something like the following in your bash terminal:

```
ln -is `Rscript -e 'cat(system.file("exec/writeprofile", package="qlcData"))'` ~/bin
```

From within R your can also use the following (again, optionally changing the linked-to directory from `~/bin` to anything more suitable on your system):

```
file.symlink(system.file("exec/writeprofile", package="qlcData"), "~/bin")
```

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Moran & Cysouw (2018) "The Unicode cookbook for linguists". Language Science Press. <doi:10.5281/zenodo.1296780>.

See Also

`tokenize`

Examples

```
# produce statistics, showing two different kinds of "A"s in Unicode.
# look at the output of "example" in the console to get the point!
(example <- "\u0041\u0391\u0410")
write.profile(example)

# note the differences. Again, look at the example in the console!
(example <- "\u00d9\u00da\u00db\u0055\u0300\u0055\u0301\u0055\u0302")
# default settings
write.profile(example)
# split according to unicode codepoints
write.profile(example, sep = "")
# after NFC normalization unicode codepoints have changed
write.profile(example, normalize = "NFC", sep = "")
# NFD normalization gives yet another structure of the codepoints
write.profile(example, normalize = "NFD", sep = "")
# note that NFC and NFD normalization are identical under unicode character definitions!
```

```
write.profile(example, normalize = "NFD")
write.profile(example, normalize = "NFC")
```

`write.recoding` *Reading and writing of recoding files.*

Description

Nominal data (‘categorical data’) are data that consist of attributes, and each attribute consists of various discrete values (‘types’). The different values that are distinguished in comparative linguistics are mostly open to debate, and different scholars like to make different decisions as to the definition of values. The `recode` function allows for an easy and transparent way to specify a recoding of an existing dataset. The current functions help with the preparations and usage of recoding specifications.

Usage

```
write.recoding(data, attributes = NULL, all.options = FALSE, file = NULL)
read.recoding(recoding, file = NULL, data = NULL)
```

Arguments

`data`	a data frame with nominal data, attributes as columns, observations as rows. Optionally a single column can be supplied. In that case the argument `attributes` can be left unspecified.
`recoding`	a `recoding` data structure, specifying the decisions of the recoding. It can also be a path to a file containing the specifications in YAML format. See Details.
`attributes`	a list of attributes to be recoded. Vectors (as elements of the list) are possible to specify combinations of attributes to be recoded as a single complex attribute. When `NULL`, then all attributes are included individually.
`all.options`	For combinations of attributes: should all theoretical interactions of the attributes be considered (TRUE) or should only the actually existing combinations in the data be presented (FALSE, by default) in the recoding?
`file`	file in which the recoding should be written. The recoding template is by default written to a file in YAML format. When `file=NULL`, the template is not converted to YAML, but returned inside R as a nested list.

Details

Recoding nominal data is normally considered too complex to be performed purely within R. It is possible to do it completely within R, but it is proposed here to use an external YAML document to specify the decisions that are taken in the recoding. The typical process of recoding will be to use `write.recoding.template` to prepare a skeleton that allows for quick and easy YAML-specification of a recoding. Or a YAML-recoding is written manually using various shortcuts (see below), and `read.recoding` is used to turn it into a full-fledged recoding that can also be used to document the decisions made. The function `recode` then combines the original data with the recoding, and produces a recoded dataframe.

The `recoding data structure` in the YAML document basically consists of a list of recodings, each of which describes a new attribute, based on one or more attributes from the original data. Each new attribute is described by:

- *attribute*: the new attribute name.
- *values*: a character vector with the new value names, optionally each value name has a linkage abbreviation (the name of the name)
- *link*: a vector with length of the original number of values. Each entry specifies a link to the new value. Zero can be used for any values that should be ignored in the new attribute. Either a pure numeric vector, using the numbering of the new values, or a vector of names of the new value names. It is also possible to use the new values here without specifying the values in the preceding item.
- *recodingOf*: the name(s) of the original attribute that forms the basis of the recoding. If there are multiple attributes listed, then the new attribute will be a combination of the original attributes.
- *OriginalFrequencies*: a character vector with the value names from the original attribute and their frequency of occurrence. These are only added to the template to make it easier to specify the recoding. In the actual recoding the listing in this file will be ignored. It is important to keep the ordering as specified, otherwise the linking will be wrong. The ordering of the values follows the result of `levels`, which is determined by the current locale.

For writing recodings by hand, there are various shortcuts allowed:

- the names `attributes`, `values`, etc. can be abbreviated. The first letter should be sufficient.
- the `recodingOf` can be the full name of the attribute in the original data, or simply a number of the column in the data frame.
- the specification of `attribute` and `values` can be left out, although the result will be uninformative names like 'Att1' and 'Val1'.
- it is also possible to add an item `doNotRecode` with a vector of original attribute names (or column numbers). These original attributes will then be included unchanged in the recoded data table.

A minimal recoding consist thus of a specification of `recodingOf` and `link`. Without `link` nothing will be recoded. Omitting `recodingOf` will lead to an error.

There is a vignette available with detailed information about the process of recoding, check `recoding nominal data`.

Value

`write.recoding.template` by default (when `yaml=TRUE`) writes a YAML structure to the specified file. When `yaml=FALSE` the same structure is returned inside R as a nested list.

`read.recoding` either reads a recoding from file, or a list structure within R, and cleans up all the shortcuts used. The output is by default a list structure to be used in `recode`, though it is also possible to write the result to a YAML-file (when `file` is specified). When `data` is specified, the output will be embelished with all the original names from the original data, which makes for an even better documentation of the recoding.

Author(s)

Michael Cysouw <cysouw@mac.com>

References

Cysouw, Michael, Jeffrey Craig Good, Mihai Albu and Hans-Jörg Bibiko. 2005. Can GOLD "cope" with WALS? Retrofitting an ontology onto the World Atlas of Language Structures. *Proceedings of E-MELD Workshop 2005*, `https://web.archive.org/web/20221007002846/https://emeld.org/workshop/2005/papers/good-paper.pdf`

See Also

The World Atlas of Language Structure (WALS) contains typical data that most people would very much like to recode before using for further analysis. See Cysouw et al. 2005 for a discussion of various issues surrounding the WALS data.

Index

∗ **datasets**
    glottolog, 7
∗ **package**
    qlcData-package, 2
.expandValues, 3

asPhylo, 3, *6*

collapse.singles, *4*

FromDataFrameNetwork, *8*

getTree, *4*, 5, *8*
glottolog, *6*, 7
gsub, *15*

join.align *(*join_align*)*, 9
join_align, 9

multi2di, *4*

pass_align, 10
phylo, *4*

qlcData *(*qlcData-package*)*, 2
qlcData-package, 2

read.csv, *13*
read.profile, *2*
read.profile *(*write.profile*)*, 18
read.recoding, *2*, *13*
read.recoding *(*write.recoding*)*, 21
read_align, *2*, *9*, 11
recode, *2*, 12, *21*

stri_opts_fixed, *17*
stri_opts_regex, *17*

tokenize, *2*, *10*, 14, *19, 20*
ToNewick, *4*

write.profile, *2*, *18*, 18
write.recoding, *13*, 21